Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 10: this Friday @ 11:59 pm
  - Assignment 3: Friday 3/31 @ 11:59 pm
    - Support video and slides on Canvas
  - Lab 9: Tuesday 4/4 @ 11:59 pm
  - Assignment 4: Friday 4/14 @ 11:59 pm
    - Support video and slides on Canvas

Previous lecture

- Repetitive Minimum Problem
  - Priority Queue ADT

This Lecture

- Heap implementation of the Priority Queue ADT

Repetitive Highest Priority Problem

- Input:
  - a (large) dynamic set of data items
    - each item has a **priority**
  - a *stream* of zero or more of following operations
    - **Find a highest** priority item
    - **Insert**
    - **Remove** a highest priority item
- Examples
  - Selection sort
    - Repeatedly, **remove** a minimum item from unsorted portion
  - Huffman trie construction
    - **remove** two minimum trees

Let's create an ADT!

- The ADT Priority Queue (PQ)

- Primary operations of the PQ:
  - Insert
  - Find item with highest priority
    - e.g., findMin() or findMax()
  - Remove an item with highest priority
    - e.g., removeMin() or removeMax()

What are possible implementations of the PQ ADT?

	findMin	removeMin	insert
Unsorted Array	O(n)	O(n)	O(1)
Sorted Array	O(1)	O(1)	O(n)
Red-Black BST	O(log n)	O(log n)	O(log n)

Which implementation should we choose?

- The best implementation may not be obvious
  - operations have different runtimes
  - Depends on the application
- Compare **amortized runtimes** over a sequence of operations

	findMin	removeMin	insert
Unsorted Array	O(n)	O(n)	O(1)
Sorted Array	O(1)	O(1)	O(n)
Red-Black BST	O(log n)	O(log n)	O(log n)

Amortized Runtime

Given a set of operations performed by the application:

$$\text{Amortized runtime} = \frac{\text{Total runtime of a sequence of operations}}{\#\text{operations}}$$

Example: Huffman Trie Construction

- K-1 iterations; each iteration: 2 removeMin and 1 insert
  - K: # unique characters
- Unsorted Array: $(K-1) * [2 * K + 1 * 1] = O(K^2)$
- Sorted Array: $(K-1)*[2 * 1 + 1 * K] = O(K^2)$
- Balanced BST: $(K-1)*[2 * \log K + 1 * \log K] = O(K \log K)$

Is a BST overkill to implement ADT PQ?

- Balanced BST: ***log n*** time for all operations
- Can we do findMinimum in less time?
- BST is efficient in finding *any* item
- findMinimum only needs the highest priority item
  - Can we take advantage of this?
    - Yes!

The heap

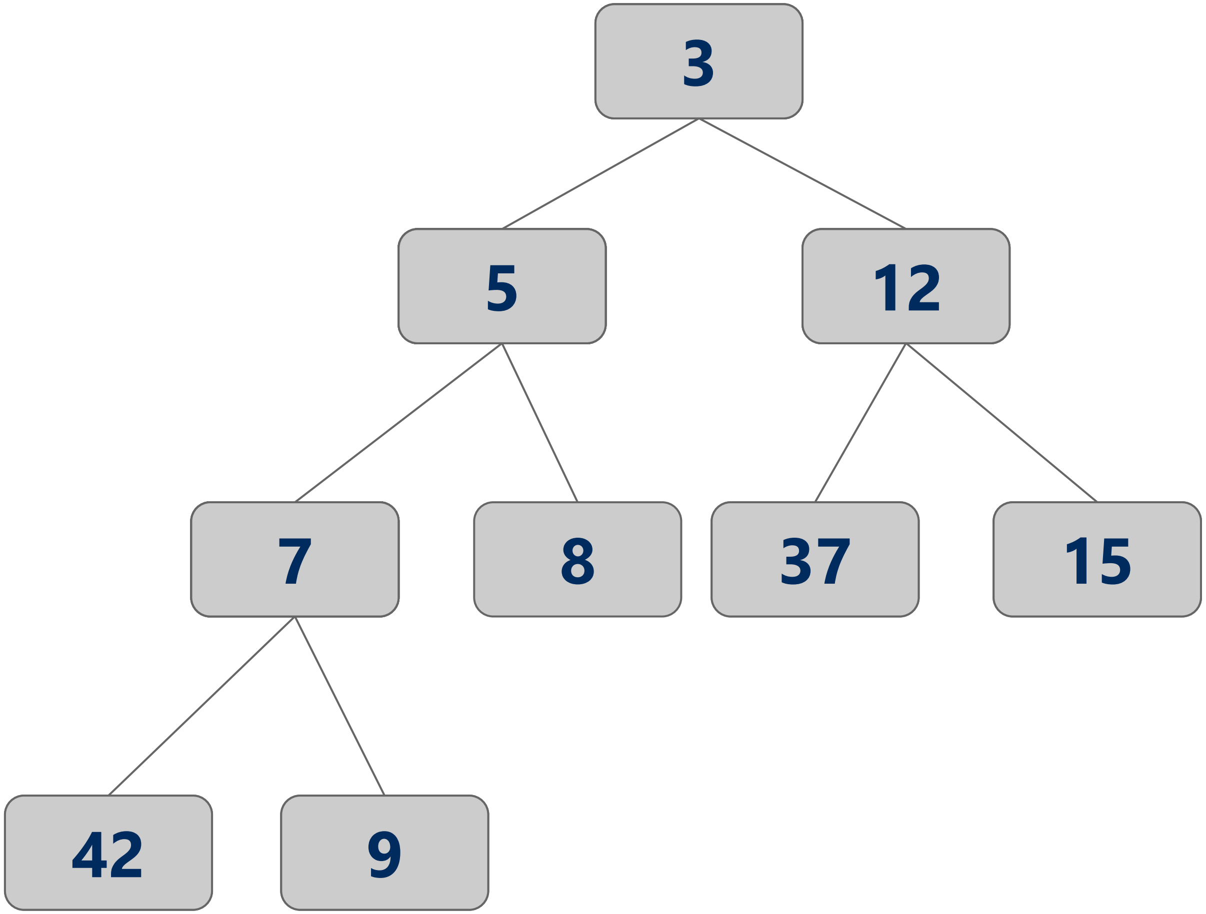
- A heap is **complete** binary tree such that for each node T in the tree:
    - T.item is of a higher priority than T.right_child.item
    - T.item is of a higher priority than T.left_child.item

- It does not matter how T.left_child.item relates to T.right_child.item
    - This is a relaxation of the approach taken by a BST

The *heap property*

Min Heap Example

- In a Min Heap, a highest priority item is a minimum item



Heap PQ runtimes

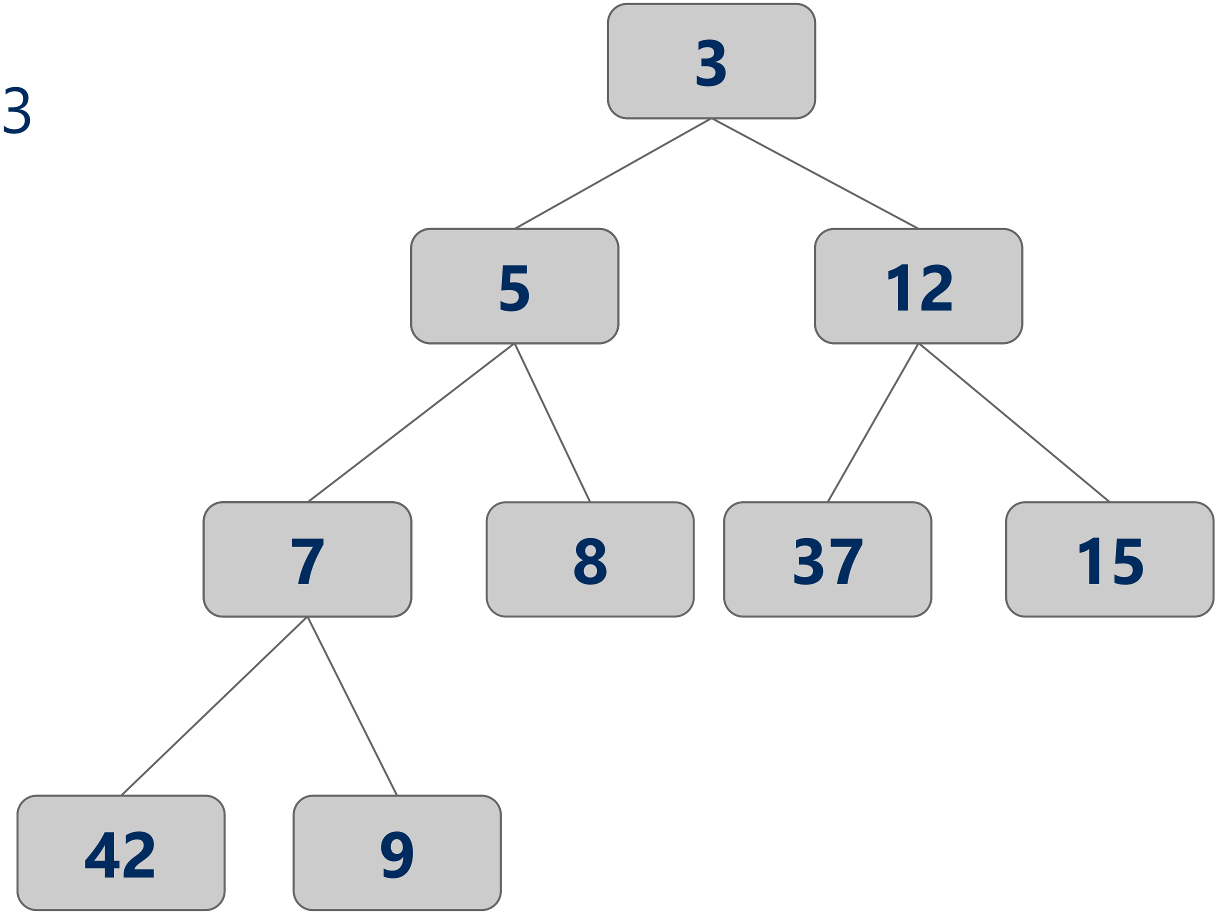
- Find is easy
  - Simply the root of the tree
    - $\Theta(1)$
- Remove and insert are not quite so trivial
  - The tree is modified and the heap property must be maintained

Heap insert

- Add the inserted item at the **next available** leaf
    - Last level of a Complete Binary Tree fills up from left to right
- Push the new item **up** the tree until heap property established

Min heap insert

Insert:
7, 42, 37, 5, 8, 15, 12, 9, 3

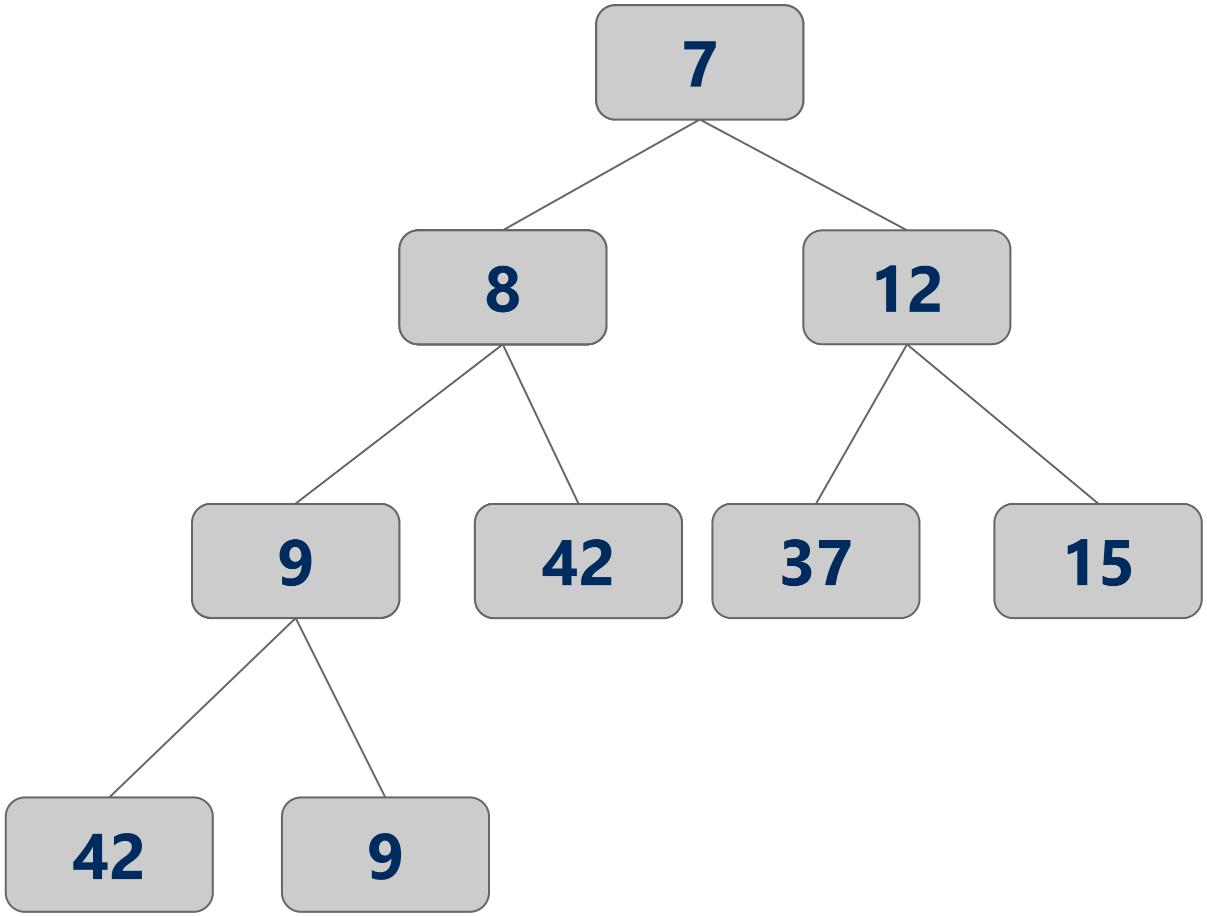


Heap remove

- Tricky to delete root…
- Instead, overwrite the root with item from the last leaf
  - then delete the last leaf
- The new root may violate the heap property…
  - push the new root **down** the tree until heap property estabished

Min heap removal

NO!



Heap runtimes

- Find
  - $\Theta(1)$
- Insert and remove
  - Height of a complete binary tree is log n
  - At most, upheap and downheap operations traverse tree height
  - Hence, insert and remove are $\Theta(\log n)$
  - Constant factors are smaller than in RB-BST because heap is simpler

What are possible implementations of the ADT PQ?

	findMin	removeMin	insert
Unsorted Array	O(n)	O(n)	O(1)
Sorted Array	O(1)	O(1)	O(n)
Red-Black BST	O(log n)	O(log n)	O(log n)
Heap	O(1)	O(log n)	O(log n)

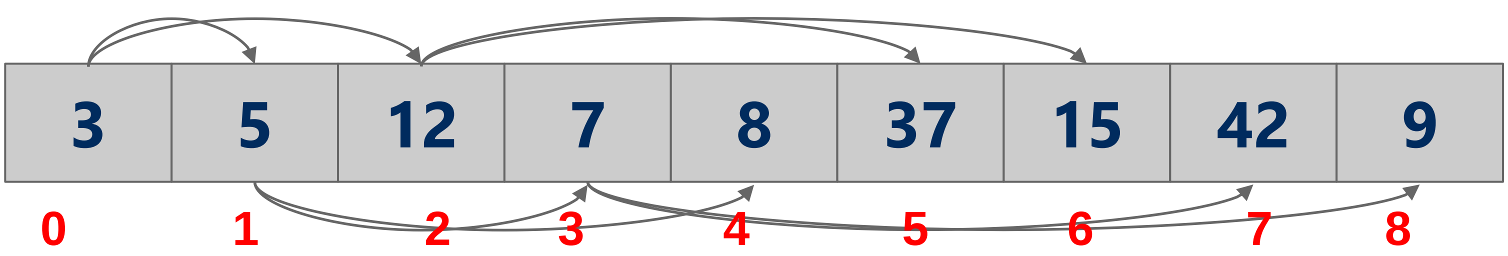
Heap implementation

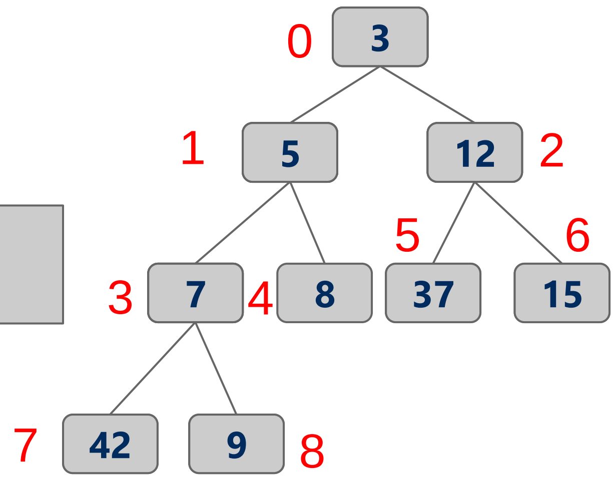
- **Linked**: tree nodes like for BinaryTree
  - overhead for dynamic node allocation
  - must have parent links
- **Array**: a heap is a complete binary tree…
  - can easily represent a complete binary tree using an array

Storing a heap in an array

- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index i
  - parent(i) = ⌊(i - 1) / 2⌋
  - left_child(i) = 2i + 1
  - right_child(i) = 2i + 2

For arrays indexed from 0





Can we turn any array into a heap?

- Yes!
- **Any array** can be thought of as a complete binary tree!
- We can change any array into a heap using an algorithm.

Heapify

- **The Heapify algorithm**

Scan through the array **right to left** starting from **rightmost non-leaf**

push item **down** the tree until heap property established

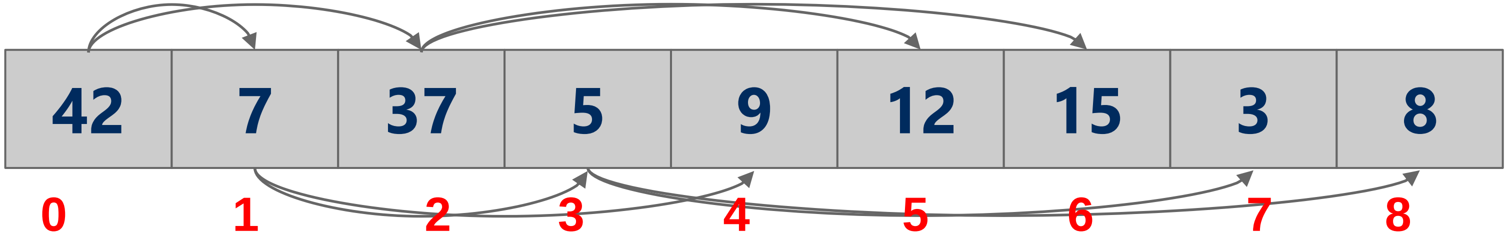
- Rightmost non-leaf is at the largest index *i* such that left_child(i) < n

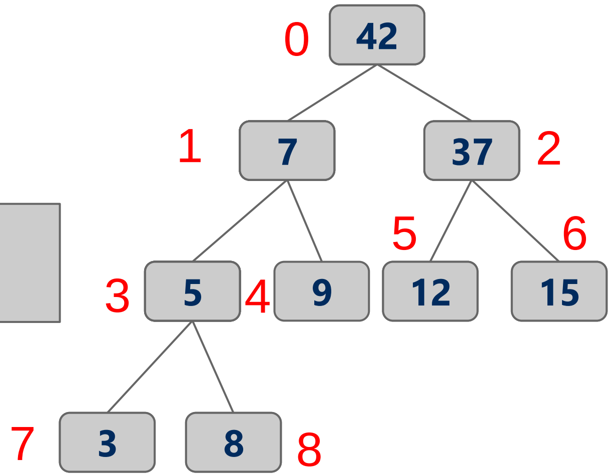
That is, 2i+1 < n

That is, i = floor((n-1)/2) if n even

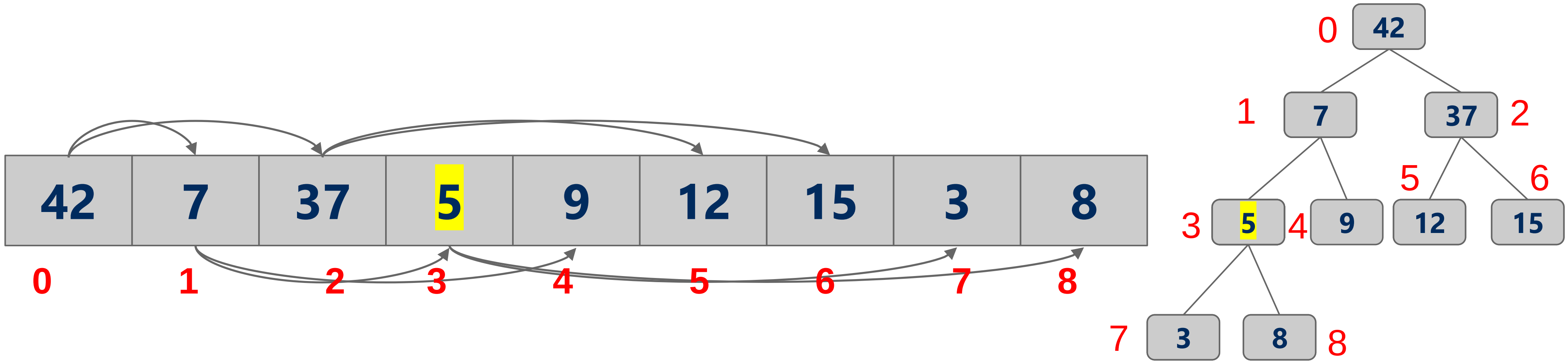
= (n-1)/2 – 1 if n odd

Heapify Example: Building a Min Heap

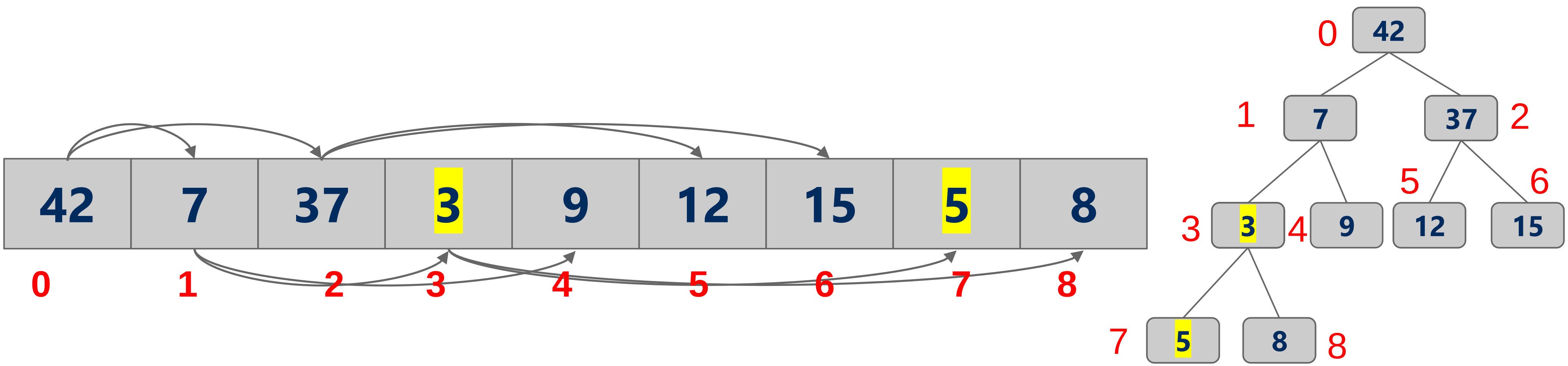




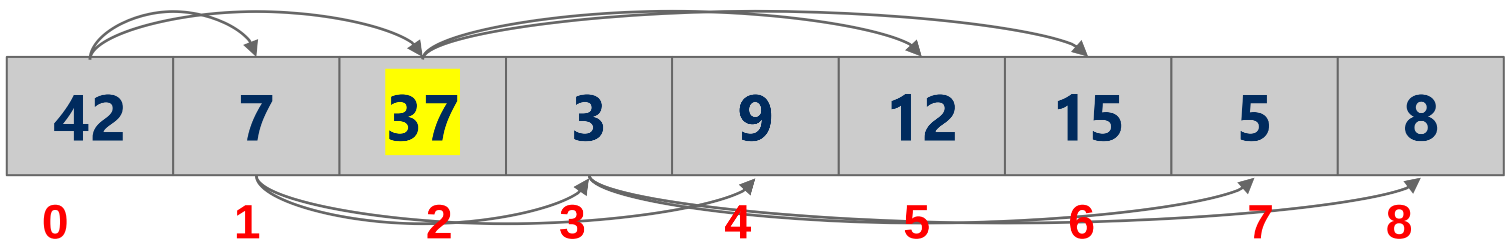
Heapify Example: Building a Min Heap

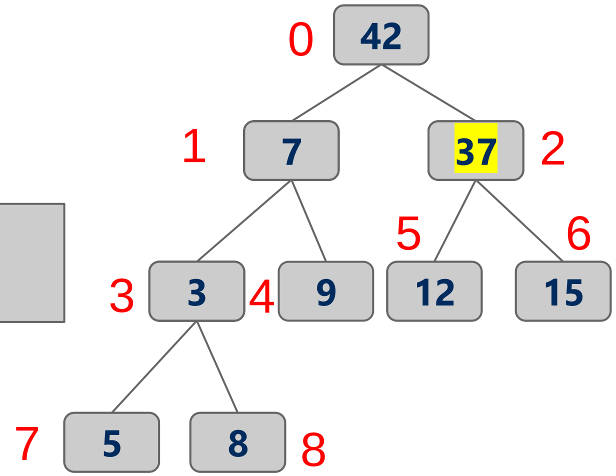


Heapify Example: Building a Min Heap

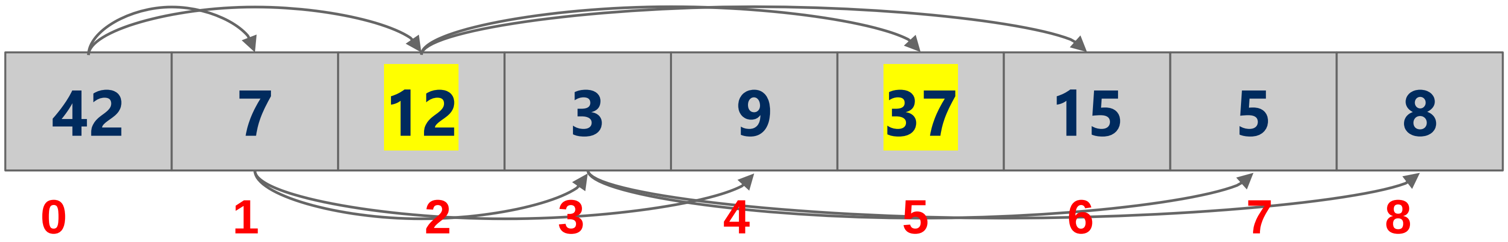


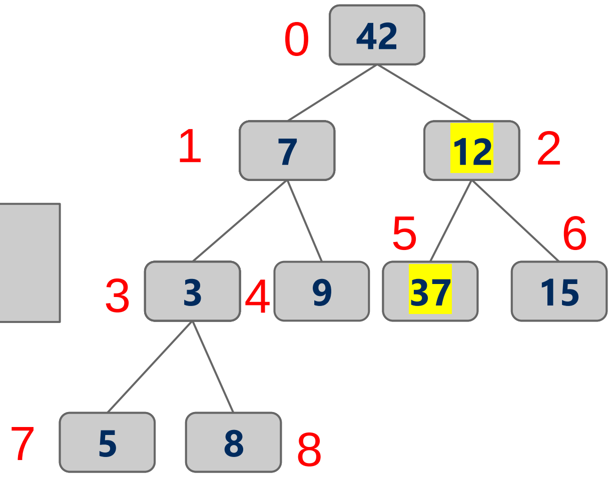
Heapify Example: Building a Min Heap



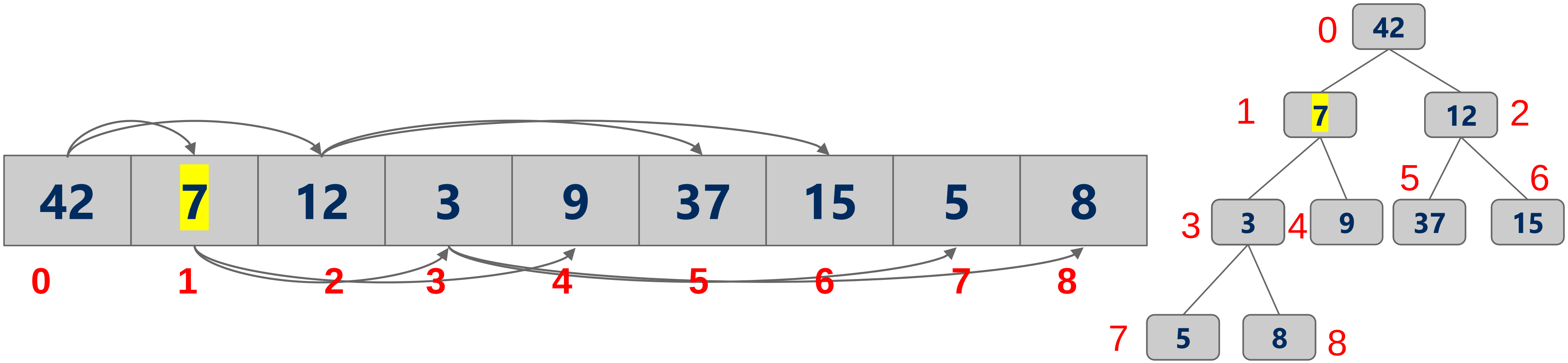


Heapify Example: Building a Min Heap

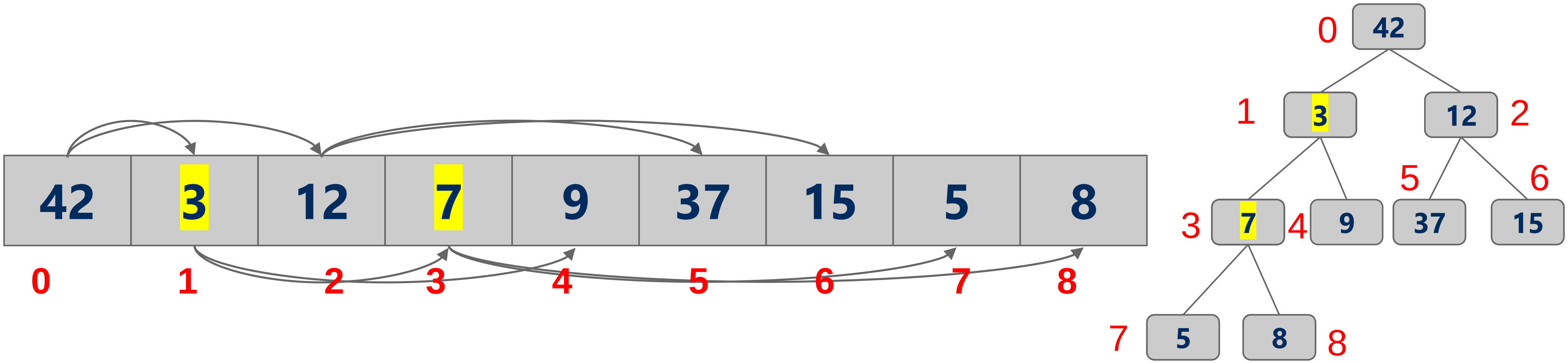




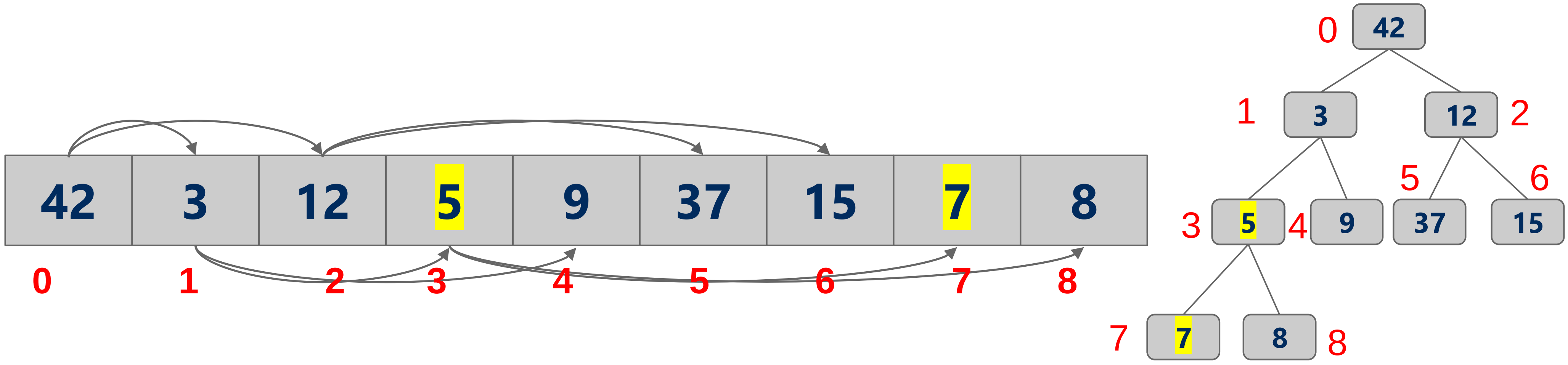
Heapify Example: Building a Min Heap



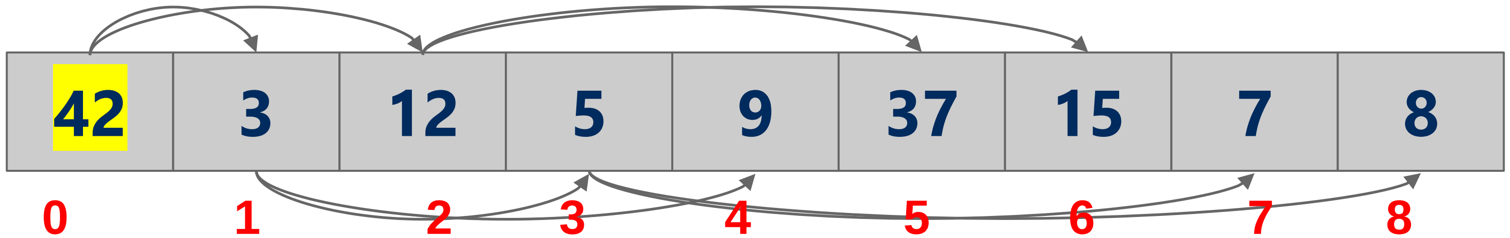
Heapify Example: Building a Min Heap

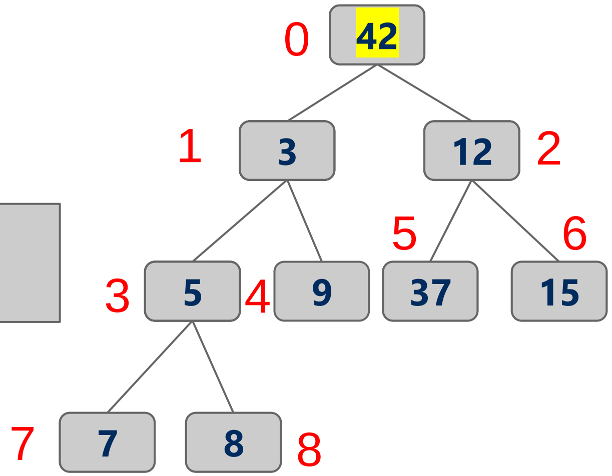


Heapify Example: Building a Min Heap

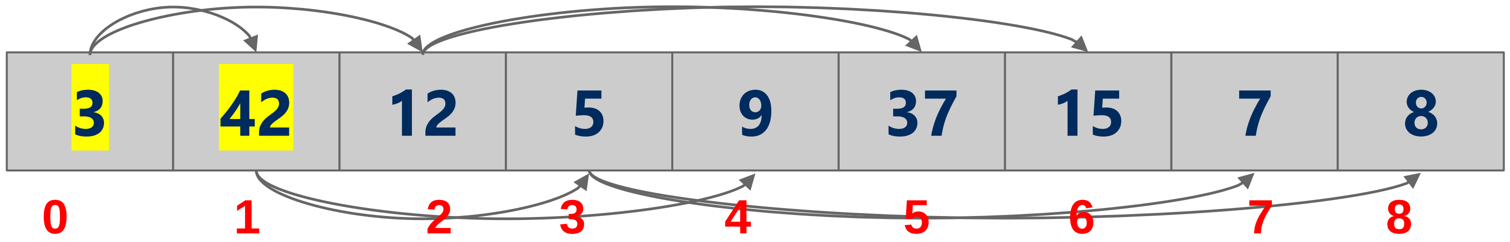


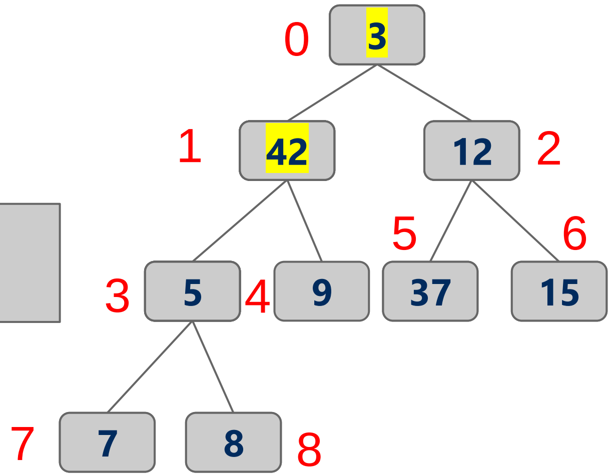
Heapify Example: Building a Min Heap



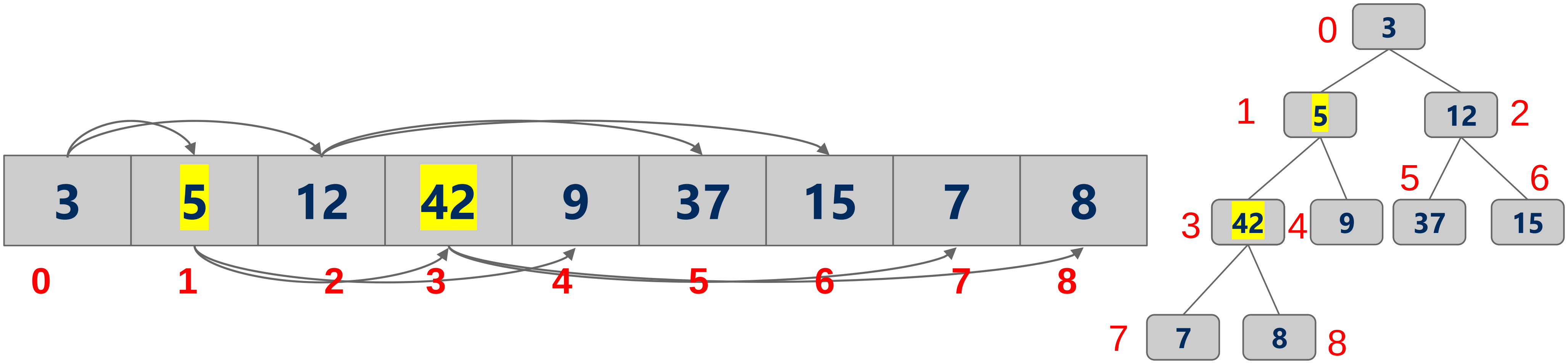


Heapify Example: Building a Min Heap

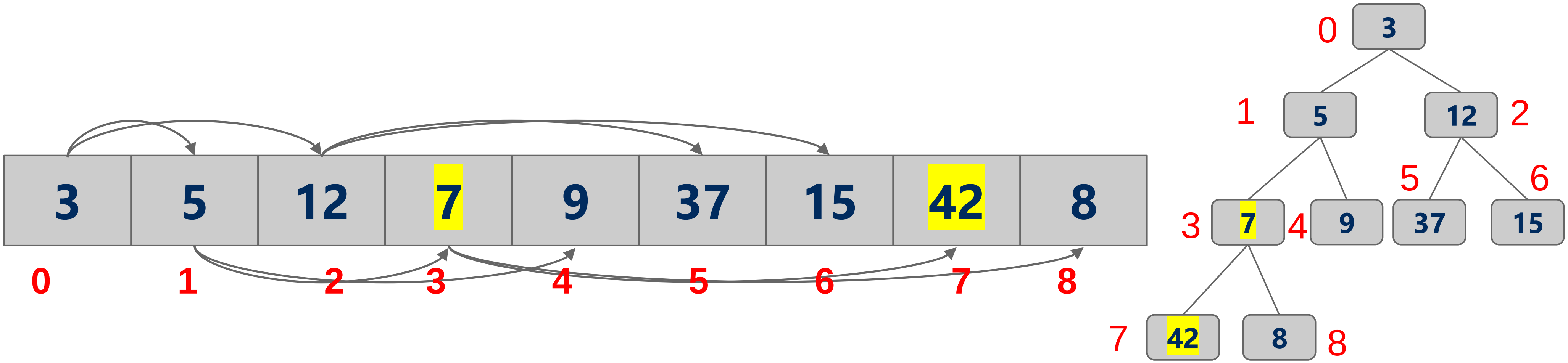




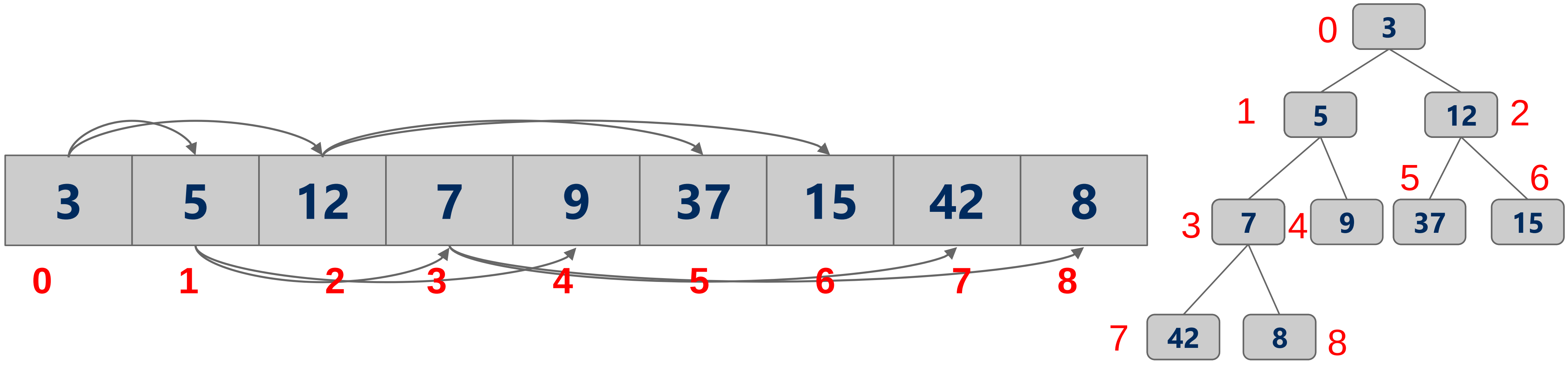
Heapify Example: Building a Min Heap



Heapify Example: Building a Min Heap



Heapify Example: Building a Min Heap

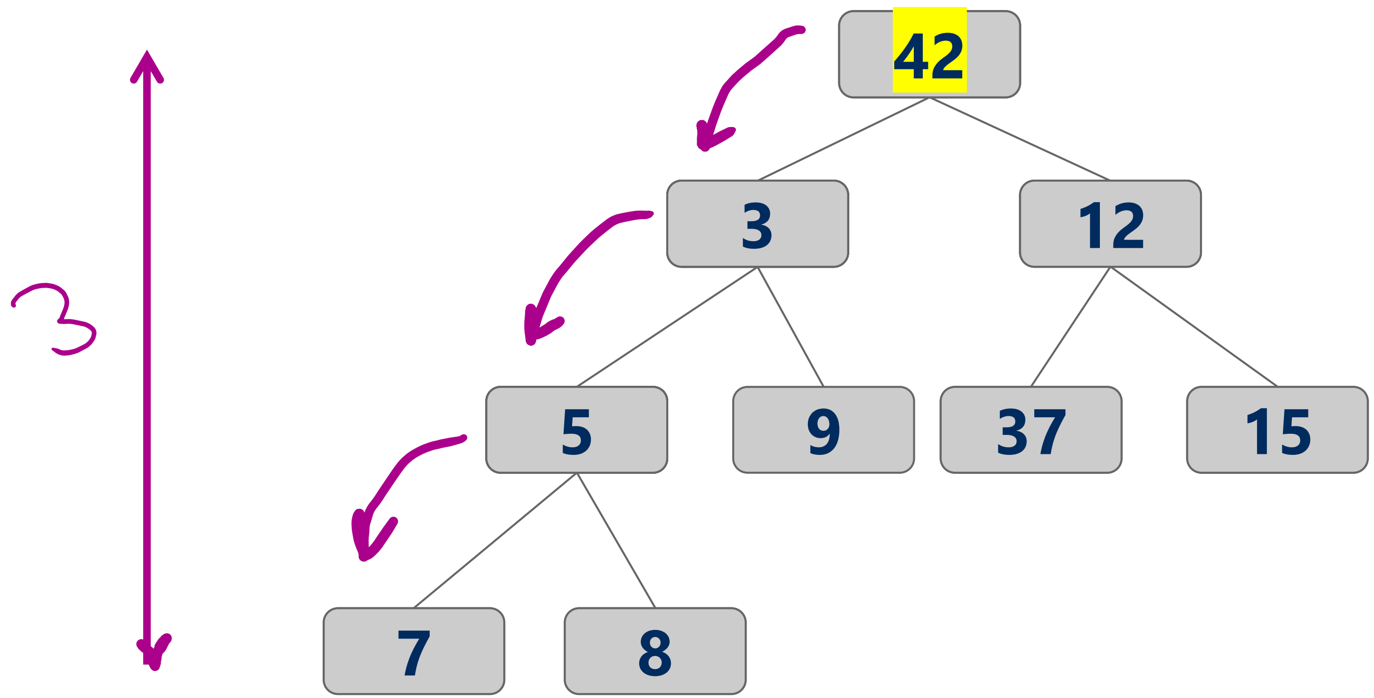


Heapify Running time

- Upper bound analysis:
  - We make about n/2 downheap operations
    - log n each
  - So, O(n log n)

Heapify Running time: A tighter analysis

- for each node, we make at most **height[node]** comparisons/swaps
- height[node] = number of edges to deepest leaf



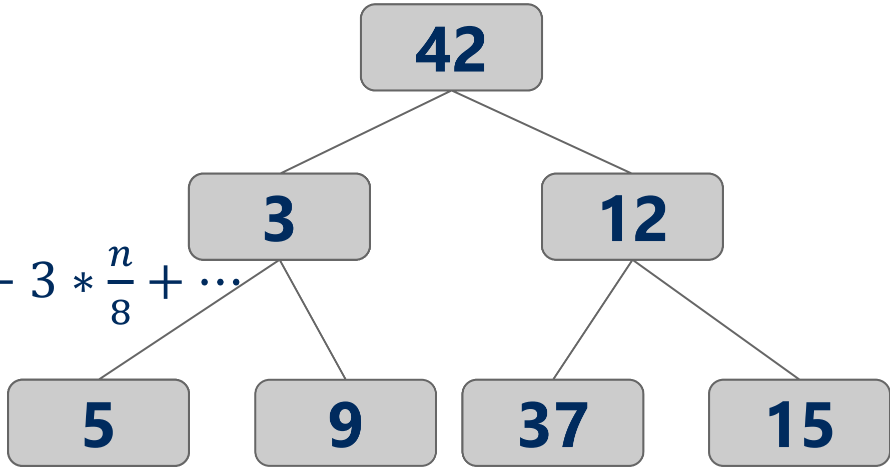
Heapify Running time: A tighter analysis

- $Runtime \leq \sum_{i=0}^{n-1} height[n]$

$$= \sum_{i=0}^{\log n} i \, * \, number\ of\ nodes\ with\ height\ i$$

- What is the number of nodes with a given height?
- Assume a full tree (worst case)
  - A node with height *i* has > 2^i nodes in its subtree including itself
  - *k* nodes with height *i* → at least $k*2^i$ nodes in their subtrees
  - *But* $k*2^i <= n \rightarrow k <= n/2^i$
- So, at most $n/2^i$ nodes exist with height *i*
- Runtime $\leq \sum_{i=0}^{\log n} i * \frac{n}{2^i} = 0 * n + \frac{n}{2} + 2 * \frac{n}{4} + 3 * \frac{n}{8} + \cdots$

$$= \quad O(largest\ term) = O(n)$$



Heap Sort

- Heapify the numbers
  - MAX heap to sort ascending
  - MIN heap to sort descending
- "Remove" the root
  - Don’t actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat

Heap sort analysis

- Runtime:
  - Worst case:
    - n log n
- In-place?
  - Yes
- Stable?
  - No

What if we need to update an item in the heap?

- A new ADT → ADT Indexable PQ
- What is the runtime to find an arbitrary item in a heap?
  - $\Theta(n)$
  - Hence, updating an item in the heap is $\Theta(n)$
- Can we improve on this?
  - Back the PQ with something other than a heap?
  - Develop a clever workaround?

Storing Objects in PQ

- What if we want to **update** an Object in the heap?
  - What is the runtime to find an arbitrary item in a heap?
    - Θ(n)
    - Hence, updating an item in the heap is Θ(n)
  - Can we improve of this?
    - Back the PQ with something other than a heap?
    - Develop a clever workaround?

Indirection

- Maintain a second data structure that maps item IDs to each item’s current position in the heap
- This creates an *indexable* PQ

Indirection example setup

- Let's say I'm shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.
- Keep objects of the following type in the heap:

```
class CardPrice implements Comparable<CardPrice>{
    public String store;
    public double price;
    public CardPrice(String s, double p) { … }
    public int compareTo(CardPrice o) {
        if (price < o.price) { return -1; }
        else if (price > o.price) { return 1; }
        else { return 0; }
    }
}
```

Indirection example

- n = new CardPrice("NE", 333.98);
- a = new CardPrice("AMZN", 339.99);
- x = new CardPrice("NCIX", 338.00);
- b = new CardPrice("BB", 349.99);

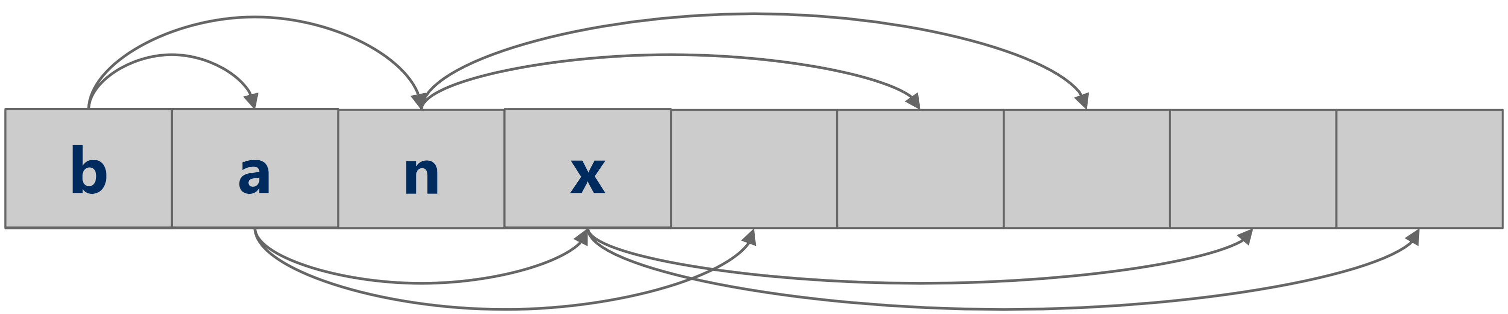
- Update price for NE: 340.00

- Update price for NCIX: 345.00

- Update price for BB: 200.00

Indirection

"NE":2
"AMZN":1
"NCIX":3
"BB":0



Indexable PQ Discussion

- how are our runtimes affected?
  - findMin, Insert, removeMin?
- space utilization?
- how should we implement the indirection?
- what are the tradeoffs?